Stochastic Computation from Serial to Parallel

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Zhiheng Wang

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Prof Kia Bazargan

December, 2017

Acknowledgements

Thanks for my adviser Prof Kia Bazargan's help though my master and Ryan Goh's mathematical support of the papers.

Dedication

To those who held me up over the years

Abstract

Stochastic Computing is a digital computation approach that operates on random bit streams to perform complex tasks with much smaller hardware footprints compared to conventional binary radix approaches. SC works based on the assumption that input bit streams are independent random sequences of 1s and 0s.

In this dissertation, both serial and parallel configuration of SC are presented in a class of complex dynamic system.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Radix encoding is the dominant form of data representation and computation used in the majority of digital systems. However, there are drawbacks to this form of encoding: long prorogation delay and area inefficiency due to the fact that data needs to be "unpacked" before computation. For example, carry chains and partial products in addition and multiplication respectively. On the other hand, in non-radix computations (e.g., stochastic computation) the circuit area could be very small and the clock period very short, compared to radix computations. Other advantages of stochastic computing include fault tolerance, and ultra-low computation power. However, the problem with non-radix computation is that it does not scale very well. The number of different values it can represent is only n where n is the number of bits, while for the radix-2 representation, it is 2^n.

Stochastic computation works on the assumption that the inputs are independent random bit streams of zeros and ones. For instance a real number $x \in [0, 1]$ is represented by the probability of being one in the bit stream with length of L, so $x = \frac{N}{L}$, where N is the number of ones in the bit stream. To compute simple multiplication of two numbers, it only requires a single AND gate shown in Fig 1.1. The ideal of computation in probability of bit stream has been discuss in [1, 2], in the past of few years, the stochastic computation become popular again due to its extremely low power and small footprint of the core computation block [3, 4]. However, generating independent random bit streams is not a trivial task, and we propose two methods to tackle this problem: one for the serial implementation (Chapter 3), and one for the parallel implementation

(Chapter 4).

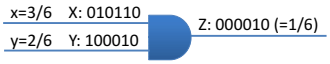

Figure 1.1: Stochastic Multiplication.

The thesis is organized as follows:

- Chapter 2 briefly introduces dynamical systems, and the science behind them, followed by the core stochastic computation scheme used in this thesis.
- In Chapter 3 the serial conventional stochastic computation and also the new architecture of the stochastic computation for dynamical systems is presented.
- Chapter 4 demonstrates the parallel implementation of the stochastic computation for dynamical systems.
- Chapter 5 shows the hardware comparison of the previous work and also the serial and parallel implementations.

Chapter 2

Dynamic System in Stochastic Computation

The Dynamic system is a board area in mathematics, here in this thesis, I only focus on the logistic map implementation in stochastic computation, specifically, the logistic map $x \to \mu x(1-x)$, which is very simple in the view of equation forms but have quite complex behavior, which is performing the $f_{n+1}(x) = f_n(x)$, where $f(x) = \mu x(1-x)$, when $n \to \infty$ the output x would converge according to the μ value, regardless of the initial value, the final converged result is shown in Fig2.1.

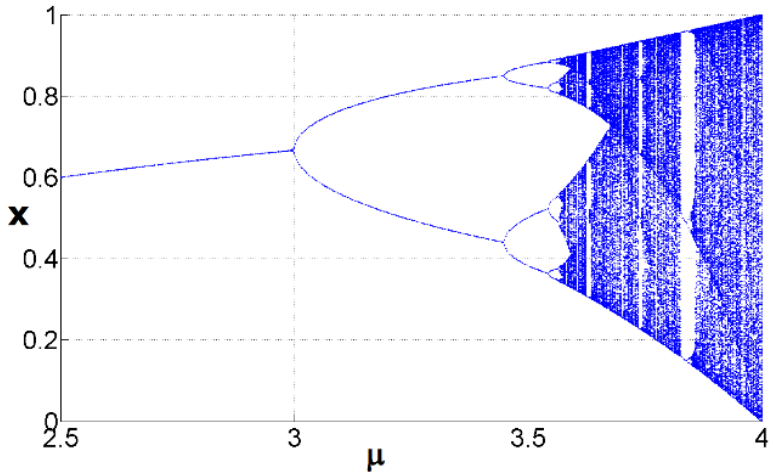

Figure 2.1: Bifurcation: logistic map convergence points as μ changes.

The analysis of such system is still a challenging problem. Reliable characterization of the statistical properties of such systems is needed and is usually done using time-domain simulations. Injecting is often necessary in these simulations to avoid long transients in temporal averages. The goal of this thesis is to analyze the complexity in dynamical systems using inherently stochastic computational tools that employ stochastic logic. [22]

For stochastic computation, which is performing the computation on the probability of the bit streams as shown in Fig 1.1 performing $P(x) \times P(y) = P(z)$, where $P(x)$ and $P(y)$ are the probabilities of the bits to be one in the x and y bit streams. The assumption of the bit streams is that they are long enough for the Central Limited Theorem to work and give us a reasonable accuracy; the longer the bit streams, the more accurate the results will get, as shown in Fig 2.2. The figure shows the mean absolute error over 10,000 runs of the multiplication.

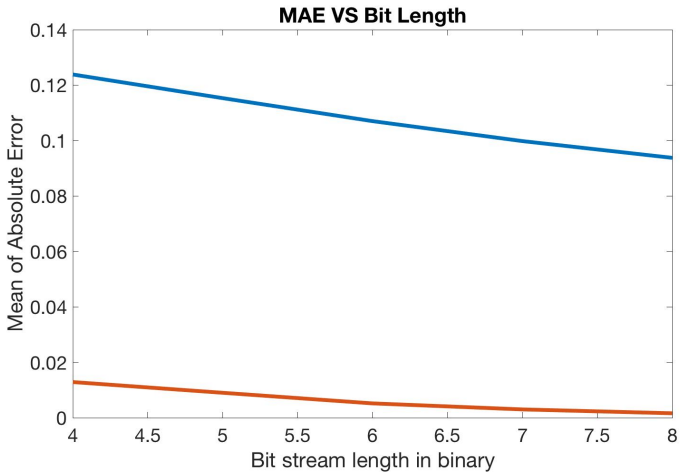

Figure 2.2: The Mean Absolute Error over 10,000 runs of the Stochastic Computation. The blue line represents the stochastic MAE and the red line shows the conventional binary MAE compared to the floating point result

The benefits of the stochastic computation compared to the conventional binary computation has provide some interesting features:

1. It has significantly reduced the size of the circuit need to implement the function, which also can enable parallelism easily. For example, in fig 1.1 shows a implementation of the multiplication for any number of bit just using a single AND gate, compared to the conventional binary needed $k \times w$ number of gates, where k is a constant greater than 2 and w is the computation width.

2. It is more robust for the noise tolerance compare to the conventional binary implementation, where the stochastic computation is not using any positional representation, so if one bit flipped, the change is only $\frac{1}{2^w}$, where w is the computation width, while in the conventional binary, the one bit noise has to be equal or greater than $\frac{1}{2^w}$

3. It has the flexibility of choosing the computation width without any change of the computation circuit itself, since the computation width for the stochastic computation only depends on the length of the bit stream.

Meanwhile, stochastic has some disadvantage compare to the binary computation, which this thesis is going to address the solution:

1. Feedback: the requirement of the stochastic computation is the independent random bit stream inputs, which is considered to be non correlated to each other, which is a challenging for the feedback system, since there is only one output bit stream to feed into multiple input bit streams.

2. Large number of independent random source: the stochastic computation requires the independent source of the random number, which is huge implementation compare to the computation block. This offsets the benefit of the area reduction in the stochastic computation.

3. High Latency: for high resolution, low noise computation, stochastic computation needs the corresponding length of the bit stream, which is 2^w where w is the binary bit width (resolution). It is increasing exponentially, which is very costly at high resolution like (16 bits).

Here in the thesis I used the iterative equation (maps) in the dynamic system as a case study to show the significant reduction in product of power and area efficient implementation.

One of the well studied example of the dynamic system is the logistic map, $x_{n+1} = \mu x_n(1 - x_n)$, where $x_i \in [0, 1]$ is a real number and i representing the number of the iterations, and μ is a constant, which value is between 0 and 4 to determine the behavior of the system when it saturated, the figure 2.1 shows the change of the μ with the saturation point(s) value. It is a very simple function but have complex behavior, since the saturation point(s) (attractors, which is the stable value after huge amount of iteration and will appear in a pattern) are entirely depended on the μ value. For example, when $1 \leq \mu \leq 3$ after $i = 200$, the system would converge to a single value where $x_{i+1} = x_i$ for $i > 200$, which is the reason we will see a single line from 2.5 to 3 in Fig 2.1, for the $3 \leq \mu \leq 3.5$, the system would finally converge to a period of 2 values and switch back and fourth, the exact value is also depended on the μ. For instance, in the case of $\mu = 3.2$, $x_{201} = 0.799$, $x_{202} = 0.513$ and then $x_{203} = 0.799$, $x_{204} = 0.513$ and so on. This is the reason that the figure has two values at the region of $3 \leq \mu \leq 3.5$. As the increasing of the μ, the attractors behavior become more complex, there are 3, 4 8 and 16 attractors region and even the chaos region where the would be no attractors.

The study of the logistic map is a demonstration of a simple function such as the $x_{n+1} = \mu x_n(1 - x_n)$ can has complex behavior in the dynamical systems. Experts in this area are still actively using it to study the statistical properties of the dynamical systems, such as densities of the states and behavior of random perturbations. The reason behind its popularity is that it is a representative for a large number of classes of map that perform period doubling and chaotic behavior. Traditionally, analyzing the dynamic systems needs huge number of experiments with random number to find the statistic distributions of points that converge, the so called attractors. As a result parallel simulation of these system in the hardware accelerator would be beneficial.

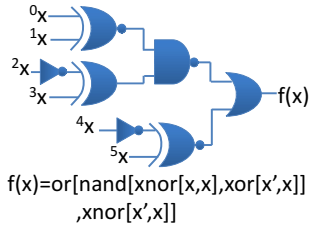

Figure 2.3: The stochastic computation block for $f(x)$

For example, in Fig 2.3 shows one stochastic implementation of one logistic map function, which function is plot in Fig 2.4, and the $f(f(x))$ function is plot in the Fig 2.5, this is a period doubling function. Fig 2.6 shows another stochastic implementation of a logistic map function, which plot is shown in Fig 2.7 and the $g(g(g(g(x))))$ plot is shown in Fig 2.8, the $g(x)$ has a period of 4 cycling. In the stochastic implementation Fig 2.3 and Fig 2.6, $AND(x, y)$ gate is the $x \times y$, $OR(x, y)$ gate is the $x + y - xy$, and $XOR(x, y)$ is the function of $x(1 - y) + y(1 - x)$ and etc. The notation of ${}^{i}x$ in Fig 2.3 and Fig 2.6 denotes the i^{th} independent bit stream, where all of the bit streams have the same probability of 1 in the bit streams, which mean all of them are same inputs.

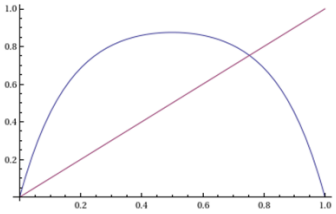

Figure 2.4: The plot of $f(x)$ as a real function $f : R \to R$(blue) and $y = x$ (red)

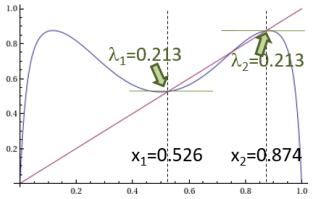

Figure 2.5: The plot of $f(f(x))$ as a real function $f(f) : R \to R$(blue) and $y = x$ (red)

Interesting fact about this logistic map is that without noise present and with low resolution of computation like 6-8 bits in the system, the behavior of period doubling and the period of 4 cycling in the Fig 2.5 and Fig 2.8 won't show.

Fig 2.4 shows the intersection between $f(x)$ and $y = x$, where the intersection is at $x = 0.7511$, which means if $f(x_i)$ gets to $f(x_i) = 0.7511$ at i^{th} iterations, it will stay at this value, $f(x_i) = 0.7511$, then $f(x_{i+1}) = f(x_{i+2}) = \ldots = 0.7511$. This means the system converges to a single unstable fixed point, which means under small perturbation, the system will end up oscillating between x_1 and x_2 shown in Fig 2.5,

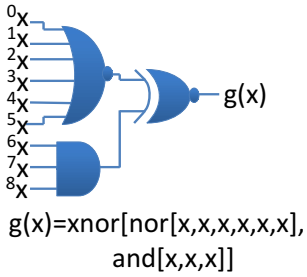

Figure 2.6: The stochastic computation block for $f(x)$

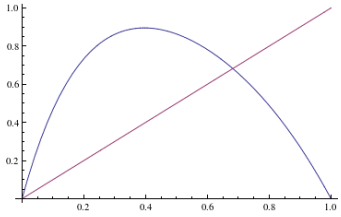

Figure 2.7: The plot of $g(x)$ as a real function $g : R \to R$(blue) and $y = x$ (red)

where are the period of 2 stable attractors in the system. The behavior of the $f(f(x))$ plot can be analyzed follow the similar procedure. Consider the point x_1, if the map generates $x_i = x_1$, then $f(x_i) = x_2$ and $f(x_2) = f(f(x_1)) = x_1$, which mean the x_1, x_2 are attractors and alternating between each other. Both x_1, x_2 are stable attractors because the derivative of $f(f(x))$ is in the range of $[-1, 1]$ at both points (λ_1, λ_2), in other words, if the system deviates from one of the attractor, it will bring the value of output back to the attractor, but might not be the same attractor but will remain in the period of cycling. Note that the analysis above is based on the assumption of perfect computation with only small normal distribution noise, which means the bit streams at all of the inputs are independent to perform as close to real number computation as possible. The correlated bit stream problem will be address in Chapter 5.

The way to get Fig 2.3 and Fig 2.6 circuit is using a greedy simulated annealing engine to synthesis the stochastic logic in both figure that implement dynamic system equation. The engine takes the circuit depth as the input and uses a balanced binary tree with the root of the tree is the output of the logistic circuit, which is a real value number $0 \leq f(x) \leq 1$, and the leaves are representing the inputs of the circuit which is

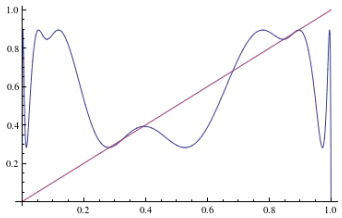

Figure 2.8: The plot of $g(g(g(g(x))))$ as a real function $g(g(g(g))) : R \to R$(blue) and $y = x$ (red)

also a real value number $0 \leq x \leq 1$. Fig 2.9 shows an example of the balanced binary tree of depth of 3 generated by the simulated annealing engine. The internal node of the tree could be wire or the logic gate, such as AND, INV, XOR , NAND, OR and etc. The annealing process randomly changes the internal node configuration to get a new circuits. The shaded regions in the Fig 2.9 are the region that the current circuit ignored because the output doesn't depend on these region. The ignored regions of the circuit are not eliminated immediately from the binary tree, because the inverter or the wire might be replaced in a later move by other gate, which could re-active the ignored regions.

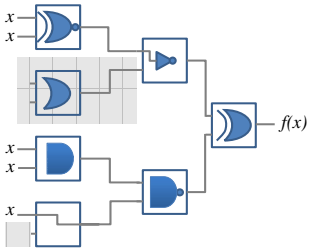

Figure 2.9: An example of the output from simulated annealing engine, a balanced binary tree with depth of 3 and shaded area is not used in this particular circuit

In order to calculate the value of the output $f(x)$ based on the given input x, first compute the internal nodes using the equation represented for the configuration of the node, for instance, $y = x$ for just a wire connection, $y = x^2$ for AND gate, and etc. The simulated annealing engine has two mode of operation:

- Curve fitting mode: given the target function $f^*(x)$, the cost function for the simulated annealing engine is the mean square error of between $f^*(x)$ and $f(x)$,

numerically calculate 100 uniformly-distributed sample points from 0 to 1.

- Function exploration mode: without given any target function, the simulated annealing engine is set to find the dynamic system behavior function. The cost function is a number of heuristics that find the interesting functions shows the property of the dynamic system. The example of such properties includes:
  - The function $f(x)$ is a n-shape function and $f(0) = 0$ and $f(1) = 0$. If the function doesn't meet with this condition, a large penalty is added to the simulated annealing cost function.
  - The number of times $f(x)$ crosses $y = x$ line has to be two. (One obvious crossing happens on the $x = 0$, we need to have another crossing so that system can have a periodic cycling attractors.)
  - The two crossing points between $f^2(x)$ and $y = x$ have to be stable. This condition can translate into $\lambda = \|f^2(x)\| < 1$ at these two crossing points. Fig 2.5 shows the two λ values. The λ is the derivative of the points where $x = f^2(x)$.
  - The number of inputs is also considered in the simulated annealing engine, because the more inputs the more area that stochastic computation will cost. For instance, in Fig 2.9 used 5 inputs.

The circuit shown in Fig 2.3 and Fig 2.6 are all the function found by the Function exploration mode. The annealing optimization will also change the depth of the circuit to change the complexity of the result function. Note that this simulated annealing engine won't necessary to provide the optimal solution for the curve fitting mode. However, it is powerfully enough to find the exploration mode to find the dynamic system behavior function for us to study the stochastic computation implementation.

The key assumption for the stochastic computation logic circuit is the independent of the input streams. As shown in the Fig 1.1 a 2-inputs AND gate can implement multiplication of two numbers only if the two input streams are generated independently. Suppose two streams are not generated independently, if one is exactly the same as the other one, that the function will become $y = x$ instead of $y = x^2$.

Chapter 3

Serial Implementation [23]

The most trivial way to break up the correlation is to use multiple independent pseudo-random number generators to produce the bit stream for each input of the stochastic logic. Fig 3.1 shows the basic architecture of the system where the stochastic computation block can be replaced with any function with feedback logic in the serial configuration. We can assume the core stochastic implementation of function $f(x)$ has K inputs. The circuit uses 2^W clock cycles to compute x_{n+1} from x_n, where W is the binary input width.

In order to compute x_{n+1} from x_n, where x_n is in the binary representation, the first step would be to generate the K 1-bit values from the binary value of x_n for the K independent bit streams. Then the stochastic computation logic for $f(x)$ would produce 0 or 1 at its output. The output value would be accumulated in the $w - bit$ sum logic. The process would continue for 2^W clock cycles, resulting in the binary value of x_{n+1} to be stored in the sum logic.

Note that for stochastic computation if in the serial configure, it is very easy to change the computation resolution which is only depends on the number of the clock cycle, no circuit changes needed. The parallel configuration is discussed in Chapter 4. Here the x_n denote the real value which is in $[0, 1]$ at the iteration n realized as the probability of being one in the 2^W length of bit stream. Each of the 2^W bits is generated in one clock cycle with the value of 0 or 1. The circuit uses the digital logic shown in the blue cloud to perform the stochastic computation of $x_{n+1} = f(x_n)$, while at the output of the stochastic logic, there is a counter or integrator denote as $\int$, which resets at zero

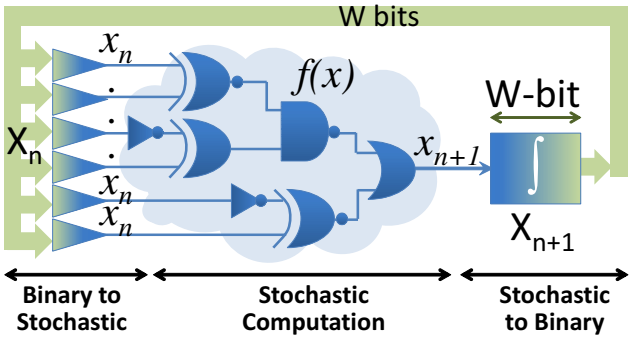

Figure 3.1: Re-randomizer based feedback. The blue cloud area is the stochastic computation logic area which works on one bit of 0 or 1 at the input. The green lines are $W-bits$ wide binary values. The initial value for the $W-bits$ counter is zero, which is on the right the stochastic to binary stage, and the $W-bits$ X_n value in the randomizer units in the left binary to stochastic stage. The output of the randomizer is 0 or 1 with the probability of being 1 as $x_n = \frac{X_n}{2^W}$. If the output of the stochastic logic is 1 that the counter at the stochastic to binary stage will increment one. After 2^W clock cycle, the counter value has the $W-bits$ binary value result, which will feedback to the inputs binary to stochastic stage.

at the beginning of each computation and produce the result at the end of computation. The whole computation takes 2^W clock cycles.

Each randomizer unit can be implemented as a $W-bit$ Linear Feedback Shift Register (LFSR) and a comparator shown in the Fig 3.2, at each clock cycle, the uniform random number generator –LFSR generates a random number which compares with the input register value X. If the value from the random number is lee than or equal to the register value, then the unit produce a one as output, otherwise a zero. It is easy to show that the output from the randomizer unit has probability of being one as input register value X for $\frac{X}{2^W}$, where X is a binary number. LFSR is a pseudo random number generator, which scan though all of the number between 0 and 2^W, where W is the width of the LFSR and the input register, and each number will only appear once in any cycle of 2^W, This unique property of the LFSR actually guarantees to have exact X ones at the output for any continues length of 2^W bit stream. For example, to get equivalent of 10 bit binary resolution the re-randomizer unit takes 1024 clock cycle to generate 1024 ones and zeros bit stream, which is coming from comparison result of

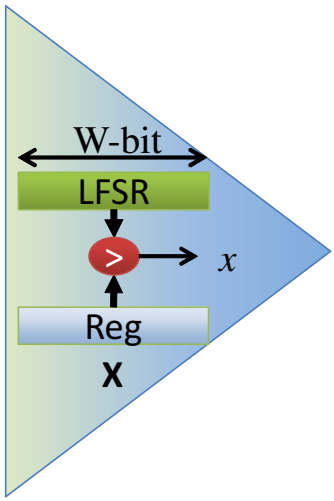

Figure 3.2: The Re-randomizer unit, which is at the binary to stochastic stage, where the input is X in binary placed in the register and output a single bit of zero or one

LFSR between Register input X, the LFSR will cycle though 1 to 1024 and repeat the pattern again, so for any 1024 cycles, the ones at the bit stream is guaranteed to be exactly X. Note that this only applies the useful configuration of the LFSR, for certain width of the LFSR the polynomial configuration is different.

The correlation between these input bit stream is highly depends on the design of the LFSR and also the initial seed of the LFSR. Especialy for LFSR the longer distance between the initial seed the smaller correlation between the output bit streams. Since the auto-correlation for the LFSR is huge for the simplest LFSR, I have proposed the LFSR4 shown in Fig 3.3 which is using three more XOR gate to have a 4 cycle number compare to the original simplest LFSR, so that the auto-correlation is small.

The majority of the area taken by the stochastic computation system is the binary to stochastic stage, which is the re-randomizer units. It would be huge benefit for the stochastic computation if we can reduce the number of the re-randomizer units without significantly hurting the accuracy of the computations, in other words, not adding correlations between the bit streams. Since the architecture shown in Fig 3.3 has little correlation between the consecutive values, we can share the bit streams generated by the re-randomizer unit and use delayed versions of their outputs to feed more than one input of stochastic logic core, which can't be achieved by simply using the LFSR

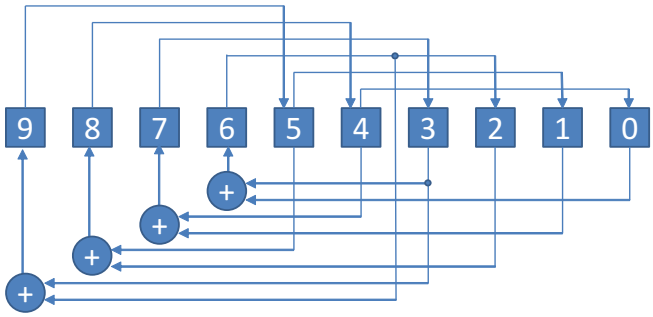

Figure 3.3: The LFSR4 random number generator for $W = 10$

with their high auto-correlation between consecutive values. For high quality random number generator would result less correlation between the consecutive values such as Mersenne Twister, which is also more expensive to implement in terms of area and delay. In Fig 3.3 shows the architecture of LFSR4 which have low correlation between the consecutive values and still maintain low cost of area and power. LFSR4 perform like a simple LFSR with improvement of cycle jump ahead. LFSR essentially is a Finite State Machine (FSM), which at each cycle it jumps to the next state, but in the LFSR4, in stead of jump one state, it jumps 4 state due the high auto-correlation between the consecutive values from the LFSR. In Fig 3.3 + denotes the XOR gate, compare to the simplest LFSR using one XOR gate, here we have 3 more XOR gate but can save multiple re-randomizer unit. We have experimentally verified that sharing one 10-bit LFSR4 for 3 to 4 inputs doesn't result in noticeable difference between what we want and the actual result. By using a wider LFSR4, one can share to more inputs, for example, in 10-bit computation using 12-bit LFSR4 can be share up to 6 inputs.

The architecture of Fig 3.1 accumulates white noise over the course of 2^W clock cycles to add the Gaussian white noise to the final value. If the system has a high signal to noise ratio, only a fraction of 2^W bots are needed to add Gaussian White noise, so we have proposed a new configuration for serial implementation of stochastic computation in Fig 3.4 to divide W bits into lower S bits and upper L bits, where $W + 1 = L + S$ and L and S can be the design parameter, which can also depend on the noise level. The computations on the L bit segment are done deterministically through a Lookup Table to map the noise free value of X_n to $f(X_n) = X_{n+1}$ and the S bit segment are

using the stochastic logic to add the Gaussian White Noise. Since we only need the S bit of stochastic computation, the number of clock cycle required is down to 2^S instead of 2^W, which is 2^{L-1} times saving. The integrator is only accumulating the lower S bits of X_{n+1} from the output of stochastic computation. For instance, $W = 10$, $L = 6$ and $S = 5$, for original stochastic computation we need 1024 clock cycle to get the final output, but in this architecture, we only need $2^5 = 32$ clock cycle for the final output. The experiement result is presented in the Chapter 5

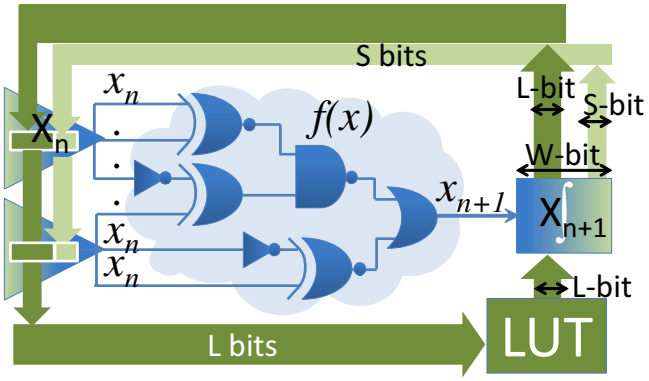

Figure 3.4: Architecture employing shared RNG and Lookup Table

Chapter 4

Parallel Implementation

The serial configuration suffers from the exponential increasing clock cycle for the final result with increasing computation width. For example, a 10-bit computation requires 1024 clock cycles for the computation. Even the enhanced architecture $L+S$ still needs to wait for an exponential number of clock cycles. The parallel configuration is a solution for the long latency, but it is not easy to implement, due to the following reasons:

1. Correlation: The correlation at the input streams has a huge impact on the final accuracy of the result. The big challenge is how to make the exponentially increasing number of input streams not correlated to each other so that the output is still valid.

2. Large area: For the serial configuration, the majority of area is taken by the re-randomizer unit. A straightforward implementation would multiply the large area of the re-randomizer unit as the degree of parallelism increases.

First, let's look at a naive way of implementing the parallel stochastic computation. In the Fig 1.1, which is shown the serial configuration of the stochastic computation, where it takes 8 clock cycle for the final output, while in Fig4.1 it is a parallel configuration, it only needs one clock cycle for the final output, it has eight parallel copies of AND gate. Fig 4.1 would also need a randomizers to generate $2^W = 8$ uncorrelated bits for the X and Y in one clock cycle, and also the output of the stochastic computation needs to transform back to the binary number.

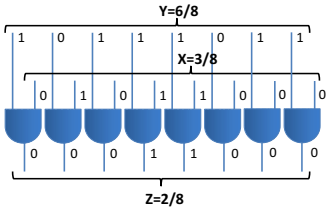

Figure 4.1: Parallel computation of the stochastic multiplication operation. Only the core stochastic logic is shown.

The circuit of Fig 4.1 would need randomizers to generate $2^W = 8$ uncorrelated bits for X and Y in one cycle. The output of the core stochastic logic needs to be aggregated into a binary number of width $W = 3$. Fig4.2 shows the randomizer and de-randomizer blocks. Note that the de-randomizer block needs to employ an adder tree to find the sum of all outputs and generate the final binary number. The adder tree and the de-randomizer unit can be pipelined for simple functions with no feedback. However, if the output of the de-randomizer is to be fed back to the randomizer units, then pipielining options would be quite limited, resulting in the clock cycle of the parallel design to be larger compared to the serial version.

An added challenge in this circuit is that the randomizer pairs that feed an AND gate should be seeded with unique random initial seeds to ensure that no AND gate is duplicating the work of another AND gate in the circuit. Furthermore, when higher resolutions are needed to compute the values (*i.e.*, larger W values), both the number of parallel copies and the bit width of the randomizers have to increase, resulting in a quadratic increase of area (as a function of W). The result is a super-linear growth of the area $\times$ delay product. These are non-trivial issues that would make it unlikely that designers adopt this method for randomizing stochastic computing. For this reason, we did not implement this design to compare to our output shuffling design, which will be discussed next. We only compared our method to the serial version.

Using randomizers and de-randomizers could be costly, especially when we need to characterize a function with higher resolutions (*e.g.*, W=10 bits). We will show in this section that randomization can be done using a much cheaper approach. To better explain our idea, let us focus on the core stochastic logic of Figure 4.1. The parallel bits in the bundle X represent a set of bits that have approximately $X \times 2^W$ ones and

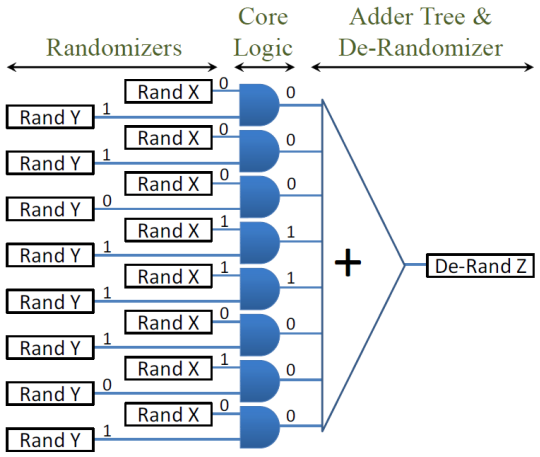

Figure 4.2: Naïve parallel multiplication method. The randomizer pairs that feed an AND gate should be seeded with unique random initial seeds to ensure that no AND gate is duplicating the work of another AND gate in the circuit. The de-randomizer needs an adder tree to accumulate all output bits into one binary number.

$2^W - X \times 2^W$ zeros[1] . The important parameter in this bundle is the number of ones vs. zeros. However, we cannot feed back the bundle Z directly as the next iteration's X or Y value directly, as will be explained next.

Let us assume we want to compute $Z = X^2Y$ using the circuit of Figure 4.1. We can potentially run the circuit in one cycle, using X and Y as the input bundles, getting a temporary value $T = X \times Y$. Then in the next cycle, we can feed X and T as input bundles to the circuit to calculate Z. Such a method would not work, because feeding T directly back as the input bundle would result in detrimentally high correlations between input bundles, invalidating the computation for Z. However, given that the output bundle T has the right ratio between 1's and 0's, we can use any permutations of the bundle to get the same value T. If we can introduce circuitry to shuffle the outputs of the AND gates, we can significantly reduce or potentially eliminate any correlations between the input bundles feeding the core stochastic logic.

Our proposed architecture for iterative stochastic functions is shown in Figure 4.3. There are N parallel copies of the core stochastic logic (in our example, $fx6$), each with

[1] If we use unique random seeds in the randomizer units, and use exactly 2^W parallel copies, we would get *exactly* $X \times 2^W$ ones in the bundle of bits.

an output register and input MUXes. Since our function has six inputs, there are six MUXes for each copy of the core logic in the figure. In our experiments we set N=2^W, but N does not necessarily have to be a power of 2. The other parameter in our design is M, which is the number of inputs of the MUXes. As we will show later, the value of M could be small, *e.g.*, M=4. All six MUXes in copy i of the core logic share the same $\log(M)$ random bits to control their select lines. The dotted lines show feedback connections from output registers to MUX inputs. These connections are fixed and randomly chosen at circuit design time. Given that our function has six inputs, the expected fanout of each output register would be six.

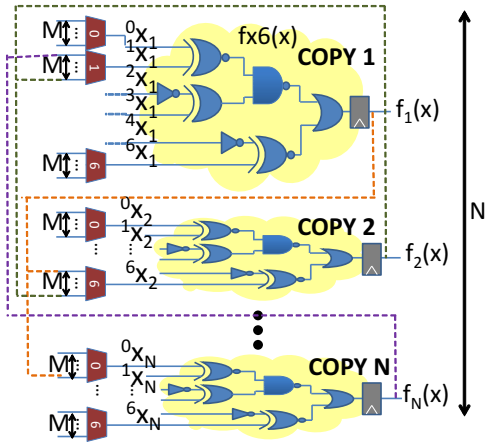

Figure 4.3: Re-randomization through output shuffling. Parameter N is the number of parallel copies (set to 2^W, W being the equivalent bit-width in binary). Parameter M is the number of inputs to the shuffling MUXes. The dotted lines show feedback connections from output registers to MUX inputs. These connections are fixed and randomly chosen at circuit design time. In each cycle, $\log(M)$ random bits are generated per copy of the core logic. All MUXes in the same copy share the $\log(M)$ random bits for their select lines.

In each cycle, we generate $N \times \log(M)$ random bits and use them to feed the select lines of the $N \times 6$ MUXes. We run the circuit for one cycle to cover one iteration of the map. The output is stored as the bundle of N bits in the output registers. To compute the next iteration of the map, we generate another $N \times \log(M)$ random select line values and repeat the process. We expect the quality of results to degrade significantly as we decrease M, but quite surprisingly, we have found that the circuit still works with M=2

and even M=1! In the case of M=1, there are *no* runtime random sources in the system, except for the initial value of X that is fed to the system through the output registers. In fact, one could argue that even the initial values of the output bits do not need to be randomly shuffled: given that the hardwired feedback connections are chosen randomly at design time, one could initialize the first K output registers to 1, and the rest to 0 to initialize the X value to K/N. We present detailed analysis of the circuit with different values of N and M in the Chapter 5.

Chapter 5

Experimental Verification and Results

The verification and the implementation results are presented in this Chapter. Since the architectures of Fig 3.4 and Fig 3.1 are similar, we only show the results of the architecture of Fig 3.1.

We used Monte-Carlo Simulations to model the behavior of the system as a Markov Chain to analyze the steady-state distributions of the stochastic computation. The steady state distribution is important because it can verify the system behavior to compare it again other system's steady state distribution on whether they have same fixed points and have the same distribution profiles. The transition probability matrix M of the system is generated by using the procedure shown in Algorithm 1. In short, the procedure runs the system using Monte-Carlo simulations, each time with a different random seed to see what $f(x_i)$ values the system is likely to get after initializing it with x_i. We store the probabilities of going from state j to state k in the j, k entry of the transition probability matrix M. If we multiply the matrix M to itself, $M_{j,k}$ would represent the probability of transitioning from state j to state k after two iterations. Similarly, M^p shows the transition probabilities between different states after p iterations of the function. Assuming that an initial state probability vector S is given with S_j stating the probability that x_0 is set to $j/2^w$ we can calculate the steady state density

vector of the system by multiplying the vector S to the matrix M^p for large values of p.

Data: Function $f(x)$, resolution 2^W, $MonteCarloPoints$
Result: Matrix $M : 2^W \times 2^W$
initialization;
forall $0 \leq i \leq 2^W$, $0 \leq j \leq 2^W$ **do**
  $M_{ij} = 0$;
end
for $i = 0$ *to* 2^W **do**
  for $j = 1$ *to* $MonteCarloPoints$ **do**
    Re-seed random number generators;
    Set $x_0 = i/2^W$;
    Run the system for 2^W micro clocks;
    y_{int} = value in the output counter;
      // real y would be $y_{int}/2^W$;
    Increment $M_{i,y_{int}}$;
  end
  Normalize the entries in row i (sum of row = 1);
end

Algorithm 1: Procedure for generating the transition probability matrix of the system modeled as a Markov Chain.

Figure 5.1 shows the results of our analysis using the function in Figure 2.3. The top row shows M for W=10, 6, 4. The graphs are similar to the shape of the function shown in Figure 2.6. As expected, binomial-like distributions are super-imposed on the function graph. Steady state densities were calculated by multiplying a uniformly distributed vector of initial states (barring states 0.0 and 1.0) to a large power of M (we used M^{2000}). The middle row of Figure 5.1 shows steady state densities, which shows all resolutions do converge to the period-2 cycle, although the W=4 case does so with less accuracy, due to its limited resolution. Finally, the bottom row of the figure shows sample time-domain simulations of the three systems. The two graphs in each plot are generated by grouping odd cycles $x_1, x_3, x_5 \ldots$ into the blue graph and even cycle values into the red graph. As W decreases, more noise is observed in the system, which is seen by switching from one stable attractor to the other in the case of W=4 (odd and even

iterations switch roles). Despite noise-induced switching between attractors, the system is still quite robust and is able to switch back to period-2 cycles. The stability of the system in low resolution modes justifies the use of a variable resolution scheme.

When comparing the steady state distribution, we can also compare the theoretical prediction of the peak width of the attractors to the actual peak width (the standard deviations of the steady state distribution) from the Markov Chain matrix with its original function: $f_a(x) = -8.778x^4 + 17.559x^3 - 14.433x^2 + 5.654x + 0.007.$, which is implemented the function shown in Fig 2.4. We used the large deviation theory developed by [16] [17][18][19] and [20] to characterize and predict the steady state distribution for this period of 2 system and how long it takes to switch the attractors like in the Fig 5.1 in bottom row for 4 bit resolution, we can clearly see the attractors has switched once, all of the theoretical prediction for the function and flat stochastic shown in Fig 3.1 and parallel configuration in Fig 4.3 implementation are all aligned with each other, thanks to Ryan's help.

In the hardware comparison table, the FX6 is referring the stochastic function shown in Fig 2.3 and FX 8 is shown in Fig 2.6, the number after FX is the number of inputs that stochastic core logic needed. We have also implemented the FX12 which is fx12(x)=XOR(AND6(x), OR6(x)), where AND6 and OR6 are AND and OR gates with 6 copies of x used as input. In order to have a fair comparison to the equivalent conventional binary implementation with our stochastic computation, we first used the curved fitting to find the polynomials in the real domain that approximate the stochastic computation function. Since our stochastic simulations were limited to a maximum of 10-bits binary resolution, it would be unfair for the conventional binary implementation to use the exact function for the FX6, FX8 and FX12. The resulting polynomials are:

$fx6 = -8.778x^4 + 17.559x^3 - 14.433x^2 + 5.654x + 0.007$

$fx8 = -1.749x^4 + 3.5x^3 - 5.712x^2 + 3.963x + 0.001$

$fx12 = -8.177x^4 + 16.354x^3 - 14.087x^2 + 5.910x + 0.002$

We used fixed-point arithmetic in the conventional implementation. Even the final resolution needs only 10 bit but the circuit needs a resolution of 16 bit for the intermediate calculations for correct steady state distribution profile, 6 out of these 16 bits are for the integer part and accommodated the signed coefficients in the range of [-14, 17]. We experimented with decreasing the number of bits in the conventional representation,

but the steady state graphs quickly degrade, hence confirming that 16 bits are needed. Fig 5.2 shows the conventional architecture. The coefficients are labeled C_i, the multipliers are labeled M_j, the accumulator as "Add" and the look-up table hardware to generate Gaussian noise as "Noise." The input value fed to the circuit is X_n, and its output is X_{n+1}, which is fed back to the input register for the next iteration. The first row of multipliers calculates the $C_1 \times x^4$ term, the second row calculates the $C_2 \times x^3$ term and so on. We avoided sharing multipliers to derive higher powers of x from lower powers (e.g., using only one multiplier to generate x^4 from x^3) because doing so would have resulted in underflow, requiring more bits. Better accuracy results are obtained when we first multiply the coefficient C_i to x, and then multiply more x to get the right term in the polynomials listed above. Since we are comparing energy ($latency \times power$) across different designs (last two columns of Table 5.1), using a more serial architecture which reuses multipliers would not result in significant changes in the total energy for the computation.

We implemented all designs in Verilog and compiled them on Artix 7 XC7K70TFBV676-1 FPGAs using the Xilinx Vivado default design flow. The chip is large enough to implement all our circuits. The architecture of Fig 5.2 is labeled "conventional", the architecture of Fig 3.1 is called "Flat stoch", the same architecture with shared LFSR4 randomizers is called "Shared LFSR", and the architecture of Figure 3.4 is called LxSy, where x and y are the number of bits used as L and S in Figure 3.4. All stochastic implementations use W=10. In the LxSy architecture we allow one bit of overlap between the S and the L parts for better accuracy (*i.e.*, the look-up table initializing the L leftmost bits, and the stochastic process accumulating the lower S bits, with a potential of bleeding into the lower bit of the L segment).

FPGA resource usage is shown under columns "LUT" (Look-up table) and "FF" (flip-flops) of Table 5.1. Note that in an FPGA, the area of the circuit is a combination of LUTs and FFs[1] . The noise generation circuit in the conventional design takes a non-trivial amount of area[2] because it has to generate Gaussian noise. The stochastic

[1] The Xilinx FPGA synthsis tool groups together LUTs and FFs into slices or CLBs (configurable logic blocks). It is common in the FPGA community to report LUTs and FFs separately and not focus on slices or CLBs.

[2] In the case of large functions such as fx6, that area would be very small – about 1% – but for smaller circuits such as those presented in Table 5.4, the Gaussian noise generation circuit would take about 10% - 20% of the total circuit area

method generates the Gaussian noise almost for free by adding up uniform noise over 2^W (or 2^S) micro cycles. The size of the look-up table containing the CDF of the normal distribution for the conventional noise generator was set to achieve similar noise characteristics to those generated by the flat stochastic design (3σ of [-0.05, 0.05] with 10 bits of resolution). Among the hybrid architectures, L7S4 shows similar noise behavior as the flat design: even though it no longer is a smooth normal distribution function, its variance is close to the σ value of the flat design. The variance of L6S5 and L5S6 are higher. Given that the stochastic implementation is very stable even with low resolutions (Figure 5.1, middle row), the fact that the resulting noise of LxSy is coarser than the Gaussian noise generated by the flat design does not seem to have an adverse effect on the quality of the steady state distribution generated by the circuit.

Table 5.1: Comparison between the conventional and the stochastic implementations of three functions with feedback

Fun	Architecture	LUT	FF	f MHz	Cycles	Delay (μs)	Delay ratio	Power (W)	Energy (latency × power)	Energy ratio
FX6	Conventional	2357	191	100	4	0.040	1.0	0.073	0.00292	1.00
	Flat Stoch.	118	96	250	1024	4.096	102.4	0.014	0.057344	19.64
	Shared LFSR	60	56	250	1024	4.096	102.4	0.010	0.04096	14.03
	L5S6	80	70	250	64	0.256	6.4	0.00396	0.001014	0.59
	L6S5	60	63	250	32	0.128	3.2	0.00294	0.000376	0.22
	L7S4	44	53	250	16	0.064	1.6	0.007	0.000623	**0.21**
FX8	Conventional	2263	185	100	4	0.040	1.0	0.152	0.00284	1.00
	Flat Stoch	142	118	250	1024	4.096	102.4	0.015	0.06144	21.63
	Shared LFSR	60	56	250	1024	4.096	102.4	0.011	0.0450	15.86
	L5S6	101	84	250	64	0.256	6.4	0.00521	0.001334	0.86
	L6S5	75	75	250	32	0.128	3.2	0.00423	0.000541	0.35
	L7S4	59	59	250	16	0.064	1.6	0.00607	0.000388	**0.25**
FX12	Conventional	2294	191	100	4	0.040	1.0	0.071	0.00284	1.00
	Flat Stoch	207	162	250	1024	4.096	102.4	0.071	0.08192	28.84
	Shared LFSR	91	82	250	1024	4.096	102.4	0.013	0.053248	18.75
	L5S6	161	112	250	64	0.256	6.4	0.00698	0.001787	1.27
	L6S5	129	97	250	32	0.128	3.2	0.00572	0.000732	0.52
	L7S4	69	82	250	16	0.064	1.6	0.010	0.00064	**0.31**

Here also an other we present the results of our experiments with different logistic-map functions. We chose three values of μ in which the logistic map exhibits period-2 cycles: $\mu = 3.175$, 3.3, and 3.4. These three functions are easy to implement in the conventional method with relatively little cost. We used the curve-fitting mode of our

Table 5.2: Comparison between different resolutions when implementing FX6.

Function	Architecture	LUT	FF	f (MHz)	Latency (cycles)	Latency (μs)	Latency ratio	Power (W)	Energy (latency × power)	Energy ratio
FX6: W = 10	Conventional	2357	191	100	4	0.040	1.0	0.073	0.00292	1.00
	Flat Stoch.	118	96	250	1024	4.096	102.4	0.014	0.057344	19.64
	Shared LFSR	60	56	250	1024	4.096	102.4	0.010	0.04096	14.03
	L5S6	80	70	250	64	0.256	6.4	0.00396	0.001014	0.59
	L6S5	60	63	250	32	0.128	3.2	0.00294	0.000376	0.22
	L7S4	44	53	250	16	0.064	1.6	0.007	0.000623	**0.21**
FX6: W = 11	Conventional	2263	185	100	4	0.040	1.0	0.152	0.00284	1.00
	Flat Stoch	142	105	250	1024	4.096	102.4	0.016	0.065536	22.14
	Shared LFSR	74	61	250	1024	4.096	102.4	0.012	0.049152	16.60
	L6S6	50	68	250	64	0.256	6.4	0.008	0.002048	0.69
	L7S5	61	73	250	32	0.128	3.2	0.009	0.001152	0.39
	L8S4	69	64	250	16	0.064	1.6	0.009	0.00576	**0.19**
FX6: W = 12	Conventional	2567	213	100	4	0.040	1.0	0.077	0.00308	1.00
	Flat Stoch	155	114	250	1024	4.096	102.4	0.016	0.065536	21.28
	Shared LFSR	88	66	250	1024	4.096	102.4	0.015	0.06144	19.94
	L7S6	71	84	250	64	0.256	6.4	0.009	0.002304	0.75
	L8S5	77	75	250	32	0.128	3.2	0.009	0.001152	0.37
	L9S4	105	66	250	16	0.064	1.6	0.011	0.000704	**0.23**

annealing algorithm to synthesize the stochastic versions of these functions.

Table 5.3 lists the target functions with the μ parameter (uxxxx_conv), and the curve-fitted stochastic implementations approximating it (uxxxx_stch). We list the function equation, along with the two fixed points and the mean-squared approximation error. It can be seen that our annealing algorithm was able to find close approximations to these functions. Table 5.4 compares the hardware performance metrics of the different implementations. LFSR4's were used with a maximum input sharing of 6, which seemed to be the limit for these functions. Note that the $L7S4$ implementation has similar noise properties compared to the conventional and it is the one that we have shown in bold. Other implementations of $LxSy$ are included as before to show the trade-off between noise profile and area / latency.

It can be seen that the stochastic implementation outperforms the conventional implementation in terms of energy consumption (70%, 88% and 75% for $u = 3.175$, 3.3, and 3.4 respectively). As expected, since the logistic function is less costly to implement in the conventional method compared to the functions used in Chapter 3, we see less performance improvements using the stochastic implementations in Table 5.4 compared to Table 5.1.

Table 5.3: Function parameters for the curve-fitting mode

	# inputs	function	fixed points	mean-squared error
u3175_conv		3.175 $x(1-x)$	[0.523, 0.792]	2.32e-3
u3175_stoch	6	ixor[ior[nor[x,x],x],nand[x,and[x,x]]]	[0.527, 0.779]	
u33_conv		3.3 $x(1-x)$	[0.479, 0.824]	1.44e-05
u33_stoch	12	ior[xor[nand[nor[x,x],inv[x]],inand[nand[x,x],x]],nor[xnor[ior[x,x],ior[x,x]],inv[xor[x,x]]]];	[0.486, 0.822]	
u34_conv		3.4 $x(1-x)$	[0.452, 0.842]	3.34e-5
u34_stoch	12	nand[ixnor[ior[inv[x],or[x,x]],inand[nand[x,x],x]], inand[inand[ixor[x,x],or[x,x]],inv[and[x,x]]]]	[0.459, 0.838]	

Table 5.4: Comparison of different logistic map functions (suitable for conventional implementation) to their stochastic implementations using a 10-bit resolution.

Function	Architecture	LUT	FF	f (MHz)	Latency (cycles)	Latency (μs)	Latency ratio	Power (W)	Energy (latency × power)	Energy ratio
u3175_conv	Conventional	220	53	100	4	0.040	1.0	0.016	0.00064	1.00
u3175_stoch	Flat Stoch	125	96	250	1024	4.096	102.4	0.012	0.049152	72.28
	Shared LFSR	70	60	250	1024	4.096	102.4	0.006	0.024576	36.14
	L6S5	42	59	250	32	0.128	6.4	0.0084	0.001075	1.58
	L7S4	44	52	250	16	0.064	3.2	0.007	0.00448	**0.70**
	L8S3	52	45	250	8	0.032	1.6	0.007	0.00224	0.33
u33_conv	Conventional	276	53	100	4	0.040	1.0	0.018	0.00072	1.00
u33_stoch	Flat Stoch.	108	54	250	1024	4.096	102.4	0.008	0.03276	45.5
	Shared LFSR	76	78	250	1024	4.096	102.4	0.008	0.03276	45.5
	L5S6	109	108	250	64	0.256	6.4	0.011	0.00281	3.90
	L6S5	74	95	250	32	0.128	3.2	0.010	0.0013	1.80
	L7S4	69	82	250	16	0.064	1.6	0.09	0.00057	**0.79**
	L8S3	72	69	250	8	0.032	0.8	0.09	0.00029	0.40
u34_conv	Conventional	274	53	100	4	0.040	1.0	0.017	0.00068	1.00
u34_stoch	Flat Stoch	201	162	250	1024	4.096	102.4	0.016	0.065536	102.40
	Shared LFSR	51	62	250	1024	4.096	102.4	0.007	0.028672	44.80
	L6S5	74	95	250	32	0.128	6.4	0.010	0.00128	2.00
	L7S4	70	82	250	16	0.064	3.2	0.008	0.000512	**0.75**
	L8S3	48	53	250	8	0.032	1.6	0.010	0.00032	0.47

Table 5.5: Comparison between the conventional and the stochastic implementations of fx6

				ASIC				FPGA				
Function	Mux Select	Resolution	Latency	Area	Delay	Product	Product Ratio	LUT	FF	Delay	Product	Product Ratio
Conventional		8	6	28110.0	2.53	426709.8	76.5	2808	310	10	187080	21.2
Stoch	1	8	1	10703.2	0.54	**5779.7**	1	981	784	5	**8825**	1
Stoch	2	8	1	18096.3	0.54	9772.0	1.7	1619	785	5	12020	1.4
Stoch	4	8	1	33078.3	0.54	17862.2	3.2	2649	807	5	17280	1.9
Flat Stoch		8	256	834.2	0.55	117455.3	21.0	69	79	5	189440	21.4
Conventional		9	6	32196.2	2.53	488737.8	38.1	2950	339	10	197340	10.1
Stoch	1	9	1	21378.3	0.6	**12826.9**	1	2333	1554	5	**19435**	1
Stoch	2	9	1	37220.0	0.6	22332.0	1.7	3355	1555	5	24550	1.3
stoch	4	9	1	66017.4	0.6	39610.4	3.0	5421	1601	5	35110	1.8
Flat Stoch		9	512	923.0	0.6	283545.6	22.1	83	88	5	437760	22.5
Conventional		10	6	36402.2	2.53	552585.0	23.7	3589	368	10	237420	6.6
Stoch	1	10	1	43164.4	0.54	**23308.7**	1	4072	3092	5	**35820**	1
Stoch	2	10	1	74657.2	0.54	40314.8	1.7	6631	3095	5	48630	1.3
stoch	4	10	1	140507.1	0.56	78683.9	3.3	10728	3103	5	69155	1.9
Flat Stoch		10	1024	996.9	0.61	622703.6	26.7	93	97	5	972800	27
L7S4[22]		10	16	NA	NA	NA	NA	44	53	5	7760	0.21

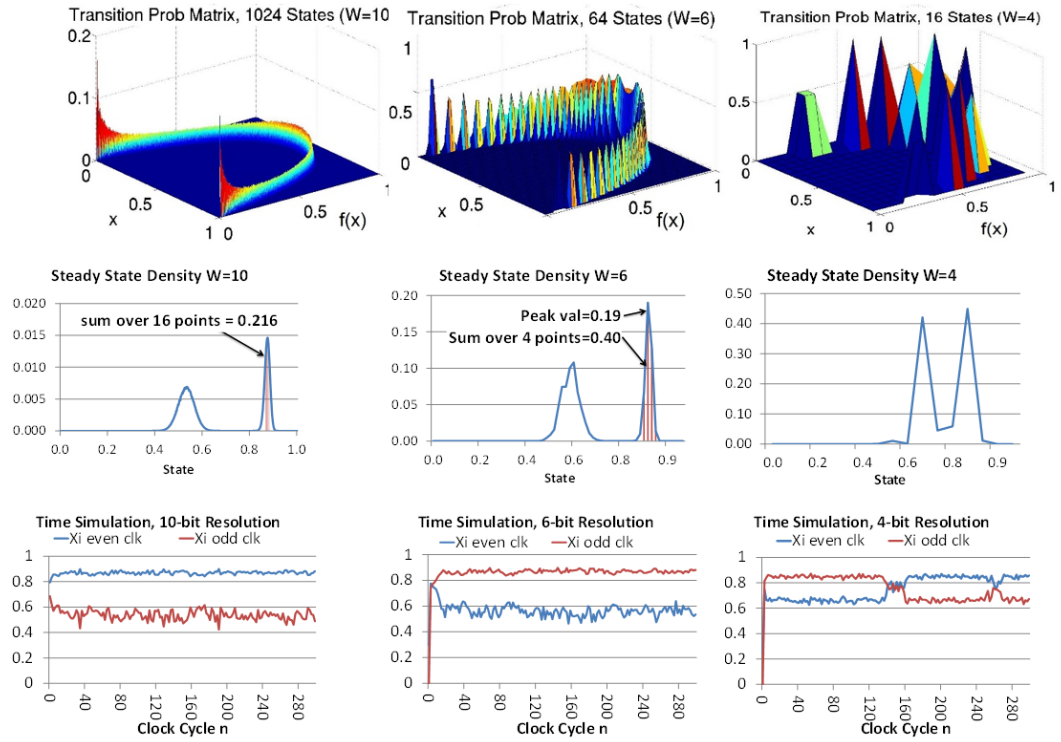

Figure 5.1: Transition probability graphs (top), steady state density (middle row) and time-domain simulation (bottom) of the circuit of Figure 3.1 for resolutions W=10 (left column), W=6 (middle) and W=4(right). The state density for W=10 is drawn using 1024 points on the x-axis. The y-axis is the probability of being in that state. The red sliver region is 16-points wide, which is of equivalent resolution as one point in the W=6 case ($2^10/16 = 2^6$). The integral of the red sliver in W=10 is 0.216, which is close to the peak value of 0.19 in W=6. Similarly, the sum of the four points near the peak in W=6 is roughly equal to the peak in W=4. In the time simulation of the W=4 case (bottom-right), the system is more susceptible to noise compared to higher resolutions, and the noise causes odd and even cycles to switch roles, but otherwise the system is stable and quickly goes back to the 2-cycle even with a limited resolution of 16 points on the x-axis.

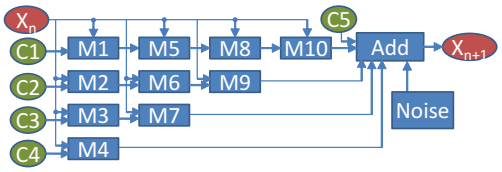

Figure 5.2: The fixed-point conventional architecture implementing the polynomial functions.

Chapter 6

Conclusion and Future Work

We have shown both serial configuration and parallel implementation of stochastic computation can be used in a class of complex dynamical systems with feedback using re-randomization of output. Serial configuration is targeted for the area limited and low power application while parallel implementation is targeted for the high performance application with reasonable low power.

In the serial configuration, results show that the simulation is stable even for low resolutions, resulting in better opportunities for optimization (e.g., noise generation, latency reduction). We also used a table lookup technique to exponentially speedup stochastic computations, resulting in better energy consumption compared to the conventional design. This is in contrast to most previous work on stochastic computing that show improved power compared to conventional binary implementations, and not energy. We verified the behavior of the stochastic implementation by comparing a number of its features such as characteristics of its steady-state distribution and mean switching time between the two attractors. The theoretical results matched very well with our experimental results.

While in the parallel implementation, results show that the system would work even if we use M=1 provided that N is large enough. We verified the behavior of the stochastic implementation by comparing a number of its features such as its steady-state density and the width of the peaks to the flat stochastic implementation.

Our future work includes optimizing stochastic computation for a more independent random source, or even remove the randomness from the system to achieve a complete

deterministic computation where the noise is bound by the quantization.

References

[1] B.R. Gaines, *Stochastic Computing Systems*, Advances in Information Systems Science J.F.Tou, ed., vol.2, chapter 2, pp.37-172, New York: Plenum, 1969.

[2] J. von Neumann, *Probabilistic logics and the synthesis of reliable organisms from unreliable components*, Automata Studies, pp. 43-98, Princeton University Press, 1956.

[3] Armin Alaghi and John P. Hayes, *Survey of Stochastic Computing*, ACM Transactions on Embedded Computing Systems (TECS), 2013.

[4] Xin Li, Weikang Qian, Marc D. Riedel, Kia Bazargan, and David J. Lilja, *A Reconfigurable Stochastic Architecture for Highly Reliable Computing*, Great Lakes Symposium (GLSVLSI), 2009.

[5] B. Brown and H. Card, *Stochastic neural computation I: Computational elements*, IEEE Trans. Comput., vol. 50, no. 9, pp. 891-905, 2001.

[6] P. Li, W. Qian, M.D. Reidel, K. Bazargan, D. Lilja, *The synthesis of linear finite state machine based stochastic computational elements,*, ASPDAC 2012.

[7] A Alaghi, C Li, JP Hayes, *Stochastic Circuits for Real-Time Image-Processing Applications*, Design Automation Conference, 2013.

[8] G. Sarkis and W. J. Gross, *Efficient Stochastic Decoding of Non-Binary LDPC Codes with Degree-Two Variable Nodes*, IEEE Communications Letters, Vol. 16, No. 3, March 2012, pp. 389-391.

[9] Jienan Chen, Jianhao Hu, *Sliding Window Method for stochastic LDPC decoder*, ISCAS 2011: 1307-1310.

[10] Naman Saraf, Kia Bazargan, David Lilja and Marc Riedel, *IIR Filters Using Stochastic Arithmetic*, Design, Automation and Test in Europe (DATE), 2014.

[11] Jienan Chen, Jianhao Hu, Shuyang Li, *Low power digital signal processing scheme via stochastic logic protection*, ISCAS 2012: 3077-3080.

[12] Peng Li, David J. Lilja, Weikang Qian, Kia Bazargan, and Marc D. Riedel, *Computation on Stochastic Bit Streams: Digital Image Processing Case Studies*, IEEE Transactions on Very Large Scale Integration (TVLSI), 2013.

[13] Demaeyer, J., and Gaspard, P. (2009), *Noise-induced escape from bifurcating attractors: Symplectic approach in the weak-noise limit*, Physical Review E, 80(3), 031147.

[14] Weikang Qian; Riedel, M.D., *The synthesis of robust polynomial arithmetic with stochastic logic*, Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE , vol., no., pp.648,653, 8-13 June 2008

[15] Weikang Qian; Xin Li; Riedel, M.D.; Bazargan, K.; Lilja, D.J, *An Architecture for Fault-Tolerant Computation with Stochastic Logic,*, Computers, IEEE Transactions on , vol.60, no.1, pp.93,105, Jan. 2011

[16] Wentzell, A. D., Freidlin, M.I., *Random Perturbations of Dynamical Systems*, Springer, 1984.

[17] Kifer, Y. *Random Perturbations of Dynamical Systems*, Birkhäuser, Boston, 1988.

[18] Kifer, Y. *A Discrete-Time Version of the Wentzell-Freidlin Theory* The Annals of Probability, vol. 18, no. 4, pp. 1676 - 1692, 1990.

[19] Hamm, A., Graham, R. *Quasipotentials for Simple Noisy Maps with Complicated Dynamics* Journal of Statistical Physics, vol. 66, no. 3, pp. 689 - 725, 1992.

[20] Reimann, P., Talkner, P. *Escape Rates for Noisy Maps* Physical Review E, vol. 51, no. 5, pp. 4105-4133, May 1995.

[21] Weber, O.; Josse, E.; Mazurier, J.; Degors, N.; Chhun, S.; Maury, P.; Lagrasta, S.; Barge, D.; Manceau, J.-P.; Haond, M. *14nm FDSOI upgraded device performance for ultra-low voltage operation* VLSI Technology (VLSI Technology), 2015 Symposium on , vol., no., pp.T168-T169, 16-18 June 2015

[22] Zhiheng Wang; Naman Saraf; Kia Bazargan; Arnd Scheel, *Randomness meets feedback: Stochastic implementation of logistic map dynamical system* Design Automation Conference (DAC), June 2015

[23] Z. Wang, R. Goh, K. Bazargan, A. Scheel and N. Saraf, *Stochastic Implementation and Analysis of Dynamical Systems Similar to the Logistic Map,* in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 25, no. 2, pp. 747-759, Feb. 2017.